Task Feasibility Analysis and Dynamic Voltage Scaling in Fault-Tolerant Real-Time Embedded Systems*

Ying Zhang and Krishnendu Chakrabarty
Department of Electrical & Computer Engineering
Duke University, Durham, NC 27708, USA
E-mail: {*yingzh, krish*}*@ee.duke.edu*

Abstract

We investigate dynamic voltage scaling (DVS) in real-time embedded systems that use checkpointing for fault tolerance. We present feasibility-of-scheduling tests for checkpointing schemes for a constant processor speed as well as for variable processor speeds. DVS is then carried out on the basis of the feasibility analysis. We incorporate practical issues such as faults during checkpointing and state restoration, rollback recovery time, memory access time and energy, and DVS overhead. Simulation results are presented for real-life checkpointing data and embedded processors.

1. Introduction

Many embedded systems in use today rely on dynamic voltage scaling (DVS) for dynamic power management (DPM). DVS is made possible by the availability of embedded processors that can dynamically scale the frequency by adjusting the operating voltage. A large number of embedded systems are also designed for real-time applications. As a result, several techniques have been proposed for low-energy, real-time task scheduling [1, 2].

Fault-tolerant computing refers to the correct execution of user programs and system software in the presence of faults. Tolerance to transient faults is typically achieved in real-time systems through on-line fault detection [3], checkpointing and rollback recovery. At each checkpoint, the system saves its state in a secure device. When a fault is detected, the system rolls back to the most recent checkpoint and resumes normal execution.

A combination of DVS and checkpointing can be used to reduce energy consumption and improve the run-time reliability of embedded systems. Moreover, a combination of the two strategies can facilitate trade-offs between energy and fault tolerance. Finally, lower processor voltages are likely to lead to lower noise margins and more transient faults. Hence DVS techniques that are tied to system-level fault tolerance are of particular interest for embedded systems.

DPM and fault tolerance for embedded real-time systems have largely been studied as separate problems in the literature. DVS techniques for power management do not consider fault tolerance [1, 2], and checkpoint placement strategies for fault tolerance do not address DPM [4, 5]. It is only recently that an attempt has been made to combine fault tolerance with DPM [6, 7, 8].

In [6], checkpointing is combined with DVS for soft real-time systems. [7] makes a number of simplifying assumptions, e.g., a task is subject to at most one fault occurrence, the processor can adjust its speed in a continuous range, and the state restoration cost is zero. In addition, faults during checkpointing and state restoration are not considered. [8] is based on similar simplifying assumptions; in addition, it uses computationally-intensive search algorithms.

In this paper, we investigate fault tolerance and DPM in hard real-time embedded systems. Our approach can handle faults during checkpointing and rollback recovery, DVS overhead, and state restoration costs. We consider job-oriented feasibility tests, in which the goal is to tolerate *k* fault occurrences for each job, as well as hyperperiod-oriented feasibility tests, in which the goal is to tolerate up to *k* fault occurrences in a hyperperiod. Following this, we extend the feasibility tests to variable-speed processors. We present a heuristic approach based on a genetic algorithm (GA) to reduce the computation cost of calculating a feasible voltage schedule. Simulation results are presented for benchmark task sets and commercial embedded processors with a discrete set of voltage/speed settings.

2. Practical issues in checkpointing and DVS

In this section, we review some practical issues in checkpointing and DVS for real-time embedded systems.

2.1 Stable storage

Checkpoints need to be saved to stable storage such that the recovery data persists through the tolerated faults [9]. Embedded systems have limited memory, and most of them do not contain a hard disk acting as nonvolatile storage. In addition to SRAM and DRAM, ROM and flash memory are used as nonvolatile storage for embedded systems. Since a ROM is a read-only device, it cannot be used for saving checkpoints.

SRAM in embedded systems is used for frequently-accessed and time-critical storage, such as caches and register files. Its typical capacity is only in the order of kilobytes, hence SRAM is of limited use for checkpointing.

DRAM is used as main memory in embedded systems, while flash memory is used for storing boot images and other

* This research was sponsored in part by DARPA, and administered by the Army Research Office under Emergent Surveillance Plexus MURI Award No. DAAD19-01-1-0504. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

1530-1591/04 $20.00 (c) 2004 IEEE

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **2004**	2. REPORT TYPE	3. DATES COVERED **-**

4. TITLE AND SUBTITLE
Task Feasibility Analysis and Dynamic Voltage Scaling in Fault-Tolerant Real-Time Embedded Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Defense Advanced Research projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR’S ACRONYM(S)

11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**6**	

non-volatile data; both have a capacity of tens of megabytes. Flash memory can be read at almost DRAM speeds, but its write access time is 10 to 100 times higher [10]. The large access time for write operations limits the use of flash memory as stable storage for short-duration real-time tasks. Therefore, DRAM is more appropriate for storing checkpoints in real-time embedded systems.

2.2 Checkpoint types and data size

Full checkpointing refers to the writing of the entire address space to stable storage during each checkpoint. In contrast, incremental checkpointing reduces data volume by writing only the pages of the address space that have been modified since the previous checkpoint [9].

For full checkpointing, only the most recent checkpoint data needs to be retained for recovery. For incremental checkpointing, old checkpoint files cannot be automatically deleted because the program's data state is spread out over many checkpoint files [11]. In resource-constrained embedded systems, it is undesirable to introduce extra hardware overhead to maintain the page table necessary for incremental checkpointing. Hence full checkpointing is more viable, despite the drawback of higher data read/write time.

The size of a checkpoint depends on the task set. For example, applications that rely on matrix operations produce megabytes of checkpoint data [11]. However, many embedded systems are targeted for real-time control in response to sensor inputs, hence data volume for such systems is limited. Furthermore, resource constraints in embedded systems limit data volume. The only source available to us in the literature that describes checkpoint size for embedded systems is [12]. The checkpoint size ranges from 0.497 KB to 2.897 KB for games on Palm handheld devices. In the absence of additional literature, we use [12] as a basis and assume that the checkpoint size is at most a few kilobytes.

2.3 Fault arrival rate

For systems that operate in harsh environments, the fault arrival rate can be as high as 10^{-2} to10^{2} per hour [13]. For example, in an orbiting satellite, the number of errors caused by protons and cosmic ray ions was measured to be as high as 35 in a 15-minute interval [13].

Prior work on checkpointing is usually based on the assumptions that no faults occur during checkpointing and that the state restoration time is zero. These assumptions are unrealistic in practice, especially for high fault arrival rates and if the checkpoint data size is not negligible.

2.4 Cost of voltage scaling

Voltage-scaling costs cannot be ignored for real-time and power-constrained embedded systems [1]. For example, the StrongArm 1100 processor takes 250 μs to switch from a 1.5V supply voltage to a 1.23V supply voltage [14]. Typically, tens of μJ of energy is needed for voltage switching. Hence for a set of short-duration real-time tasks, it is counterproductive to employ DVS. For longer-duration tasks, the consideration of DVS overheads leads to more accurate conclusions.

3. Feasibility analysis under constant speed

We are given a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic real-time tasks, where task τ_i is modeled by a tuple $\tau_i = (T_i, D_i, E_i)$, T_i is the period of τ_i, D_i is its deadline, and E_i is the worst-case execution time (WCET) of τ_i under fault-free conditions. The WCET values can be obtained using techniques described in [15]. Let the time required to store (retrieve) a checkpoint be $C_s(C_r)$. We make the following assumptions: (i) Γ is scheduled using fixed-priority methods such as the rate-monotonic scheme [16]; (ii) Γ is schedulable under fault free conditions; (iii) task τ_i has higher priority than task τ_j if $i < j$; (iv) each instance of the task is released at the beginning of the period; (v) the checkpointing intervals for the same task are equal; (vi) faults are detected as soon as they occur.

We next provide two solutions corresponding to two different fault-tolerance requirements. One is to tolerate k faults for each job (task instance), termed as job-oriented fault-tolerance; the other is to tolerate k faults within a hyperperiod (defined as the least common multiple of all the task periods [16]), termed as hyperperiod-oriented fault-tolerance. The choice of an appropriate fault tolerance criterion can be made based on the needs of the real-time application.

We first consider the case of a single job. Suppose the checkpointing interval is $\Delta = E/(m+1)$, where m is the number of checkpoints inserted equidistantly to tolerate k faults in one job. The objective here is to find the optimal checkpointing interval to minimize the worst-case response time in the presence of faults.

The total execution time of the job can be divided into three categories: effective computation (the time when the job performs real computation), checkpoint saving, and state retrieval. Based on this classification, we can further divide the occurrences of the k faults during task execution. Suppose k_1 faults occur during checkpointing saving, k_2 faults occur during checkpoint retrieval, and k_3 faults occur during effective computation, where $k = k_1 + k_2 + k_3$; see Figure 1. Whenever a fault occurs during job execution or checkpoint saving, the system rolls back to the most recent checkpoint and restores the system state. As a result, the maximum time penalty due to a fault during job execution is $\Delta+C_r$, as indicated in Figure 1(a). Similarly, the maximum time penalty due to a fault during checkpoint saving is $\Delta+C_s+C_r$, as indicated in Figure 1(b). If a fault occurs during state restoration, the system rolls back to the checkpoint and attempts to restore state, as demonstrated in Figure 1(c). Hence the maximum time penalty due to a fault during checkpoint retrieval is C_r.

The response time R for the job is composed of five terms: (i) the fault-free job execution time: E; (ii) the total time for saving m checkpoints: mC_s; (iii) the additional penalty due to k_1 faults during checkpoint saving: $k_1(\Delta+C_s+C_r)$; (iv) the penalty due to k_2 faults during state restoration: k_2C_r; (v) the penalty due to k_3 faults during job execution: $k_3(\Delta+C_r)$. Hence the response time is expressed as:

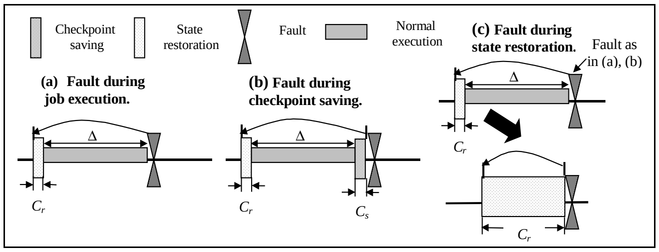

Figure 1: Illustration of fault occurrence.

$R = E + (m + k_1)C_s + (k_1 + k_3)\Delta + kC_r$. It can be seen that the worst-case response time is obtained when $k_1 = k$, and $k_2 = k_3 = 0$. This means that all k faults occur at the end of checkpoint saving. Replacing Δ with $E/(m+1)$, the worst-case response time $R_{\text{worst-case}}$ is further expressed as: $R_{\text{worst-case}}(m) = E + k(C_s + C_r) + mC_s + kE/(m+1)$. We next find the optimal value of m such that $R_{\text{worst-case}}$ is minimized. To satisfy the deadline constraint, we must have $R_{\text{worst-case}}(m) \le D$.

The minimum value of $R_{\text{worst-case}}(m)$ is obtained for $m = \sqrt{kE/C_s} - 1$. Let $m^{\Delta} = \left\|\sqrt{kE/C_s} - 1\right\|$ denote the value of m from the pair $\left\{\left\lceil\sqrt{kE/C_s} - 1\right\rceil, \left\lfloor\sqrt{kE/C_s} - 1\right\rfloor\right\}$ that minimizes $R_{\text{worst-case}}$. Furthermore, since m is a non-negative integer, we have $m_0 = \max(m^{\Delta}, 0)$. Let $f(m_0) = R_{\text{worst-case}}(m_0) - D$.

If $f(m_0) \le 0$, there exists equidistant checkpointing schemes for k-fault-tolerance, and the response time is minimum when m_0 checkpoints are inserted. If $f(m_0) > 0$, then no equidistant checkpointing schemes exists for tolerating up to k faults.

The feasibility analysis for more than one job is based on the time-demand analysis for fixed-priority, preemptive task scheduling [16]. The steps in the analysis are as following:

(1) Compute the response time R_i for τ_i according to the equation: $R_i = E_i + \sum_{h=1}^{i-1}\lceil R_i/T_h\rceil E_h$. Here T_h and E_h are the period and the execution time of a task τ_h with higher priority than τ_i. This equation can be solved by forming a recurrence relation: $R_i^{(j+1)} = E_i + \sum_{h=1}^{i-1}\lceil R_i^{(j)}/T_h\rceil E_h$. (1)

(2) The iteration is terminated either when $R_i^{(j+1)} = R_i^{(j)}$ and $R_i^{(j)} \le D_i$ for some j or when $R_i^{(j+1)} > D_i$, whichever occurs sooner. In the former case, τ_i is schedulable; in the later case, τ_i is not schedulable.

According to [16], the time complexity of the time-demand analysis for each task is $O(nR)$, where R is the ratio of the largest period to the smallest period.

3.1 Job-oriented fault-tolerance

In this case, we require that all tasks can meet their deadlines under the condition that at most k faults occur during the execution of each job.

In the worst-case scenario, the additional time due to checkpointing and recovery should be incorporated. When there are m_j equidistant checkpoints for each instance of τ_j, we have:

$$R_i = [E_i + k(C_s + C_r) + m_iC_s + kE_i/(m_i+1)]$$
$$+ \sum_{h=1}^{i-1}\lceil R_i/T_h\rceil\ [E_h + k(C_s + C_r) + m_hC_s + kE_h/(m_h+1)].$$

Let $f_i(m_i) = E_i + k(C_s + C_r) + m_iC_s + kE_i/(m_i+1)$. Then $R_i = f_i(m_i) + \sum_{h=1}^{i-1}\lceil R_i/T_h\rceil f_h(m_h)$. To minimize response time R_i, $f_i(m_i)$ must be minimized. Let $m_i^* = \max\left(\left\|\sqrt{kE_i/C_s} - 1\right\|, 0\right)$. We then employ the following relation:

$R_i^{(j+1)} = f_i(m_i^*) + \sum_{h=1}^{i-1}\left\lceil R_i^{(j)}/T_h\right\rceil\ f_h(m_h^*)$. When $R_i^{(j+1)} = R_i^{(j)}$ and $R_i^{(j)} \le D_i$ for some j, τ_i is schedulable; when $R_i^{(j+1)} > D_i$, τ_i is not schedulable. The overall time complexity of this procedure is $O(n^2R)$.

Example 1: Consider a hypothetical task set composed of two tasks: $\tau_1 = (60, 25, 7)$ and $\tau_2 = (80, 47, 8)$, and let $k = 3$, $C_s = C_r = 1$. Then $m_1^* = 4$ and $m_2^* = 4$. After applying the recurrence equation, we get the response times: $R_1 = 21.2 < 25$; $R_2 = 44.0 < 47$. Thus checkpointing is feasible for this task set if up to three faults occur during each job. Next we examine the case of $k = 4$. For this case, $m_1^* = 4$ and $m_2^* = 5$. The response times are: $R_1 = 24.6 < 25$ and $R_2 = 50.9 > 47$. As a result, checkpointing is not feasible.

3.2 Hyperperiod-oriented fault-tolerance

We require here that tasks meet their deadlines under the condition that at most k faults occur during a hyperperiod. Let $F_j = E_j/(m_j+1)$. The response time R_i for τ_i is expressed as:

$$R_i = (E_i + m_iC_s) + \sum_{h=1}^{i-1}\lceil R_i/T_h\rceil\ (E_h + m_hC_s) + k(C_s + C_r) + k\max_{1\le j\le i}\{F_j\}$$

Any time we increase the number of checkpoints for a task, all the lower-priority tasks need to be re-examined. Upper bounds on the number of checkpoints sufficient for timely completion of tasks are described in [8]. A checkpointing algorithm for off-line feasibility analysis is also described in [8]. The algorithm in [8] can be easily extended to handle faults during checkpointing and state-restoration cost by using the above expression for R_i.

4. Feasibility analysis with DVS

Here we are given a variable-speed processor equipped with l speeds $f_1 < f_2 < \ldots < f_l$. We are also given a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic real-time tasks, where task τ_i is modeled by a tuple $\tau_i = (T_i, D_i, E_i)$, T_i is the period of τ_i and D_i is its deadline ($D_i \le T_i$), and E_i is the number of computation cycles of τ_i under fault-free conditions.

We assume that processor voltage scaling does not affect the cost of checkpoint saving and state restoration. For processors with on-chip cache, we can simply consider an upper limit on the cache write-back cost. As in Section 3, we use $C_s(C_r)$ to denote the time needed for checkpoint saving (data retrieval). Let the energy cost for saving(retrieving) one checkpoint be $Eng\_C_s(Eng\_C_r)$. In addition to the assumptions in Section 3, we assume the task set Γ is schedulable under fault free conditions at the lowest speed, and a single speed switch incurs time cost of t_{ss} and energy consumption of Eng_ss.

The power consumption $P(f)$ at a clock frequency f for an embedded processor can be found in the data sheets. For a task with N single-cycle instructions, the energy consumption can be expressed as: $Eng(N, f) = P(f)*N/f$. Speed scaling can be done at the application level, i.e., all tasks for the application are assigned the same speed, or at the task level, i.e., different tasks are assigned different speeds. Speed scaling can also be carried out at the job level, i.e., different jobs for a task can have different speeds. Let $s(\tau_i) : \tau_i \rightarrow f_j$ ($1 \le i \le n$, $1 \le j \le l$) denote the speed scaling function, which maps a task τ_i to speed f_j.

Let the hyperperiod be denoted by Ht and the number of checkpoints for τ_i be denoted by m_i. The off-line feasibility analysis with DVS provides two important pieces of information: first, it provides the feasibility analysis under the worst-case scenario; second, it provides static results such as speed assignment and checkpoint interval, which can be further used for on-line adjustment during task execution.

4.1 Job-oriented fault-tolerance with DVS

The difficulty in modeling DVS cost accurately is that voltage-switching events can only be known after the schedule is obtained; hence it is not possible to characterize it during feasibility analysis. Therefore, we employ a conservative method here, which assumes that voltage switching occurs between any two consecutive jobs. If the task set can be scheduled under this conservative assumption, it is guaranteed that the task set can be scheduled under any voltage-switching scenario.

The worst-case response time for task τ_i is expressed as:

$$\begin{aligned} R_i =& [(E_i + kE_i/(m_i+1))/s(\tau_i) + k(C_s + C_r) + m_i C_s] \\ &+ \sum_{h=1}^{i-1} \lceil R_i / T_h \rceil \; [(E_h + kE_h/(m_h+1))/s(\tau_h) + k(C_s + C_r) + m_h C_s + t_{ss}] \end{aligned} \quad (2)$$

The total energy consumption during one hyperperiod is expressed as:

$$\begin{aligned} Total\_eng =& \sum_{i=1}^{n} Ht/T_i [Eng(E_i + kE_i/(m_i+1), s(\tau_i)) \\ &+ k(Eng\_C_s + Eng\_C_r) + m_i Eng\_C_s + Eng\_ss] \end{aligned} \quad (3)$$

To minimize all response times, we must have: $m_i^* = \max(\left\| \sqrt{kE_i/(s(\tau_i)C_s)} - 1 \right\|, 0), 1 \le i \le n$. As a feasibility test, we employ the recurrence equation as follows:

$$\begin{aligned} R_i^{(j+1)} =& [(E_i + kE_i/(m_i^*+1))/s(\tau_i) + k(C_s + C_r) + m_i^* C_s] \\ &+ \sum_{h=1}^{i-1} \lceil R_i^{(j)} / T_h \rceil \; [(E_h + kE_h/(m_h^*+1))/s(\tau_h) + k(C_s + C_r) + m_h^* C_s + t_{ss}] \end{aligned}$$

If $R_i^{(j+1)} = R_i^{(j)}$ and $R_i^{(j)} \le D_i$ for some j, τ_i is schedulable; if $R_i^{(j+1)} > D_i$, τ_i is not schedulable.

Since the optimal number of checkpoints depends on the speed assignment, we first need to choose appropriate processor speeds. After that we can calculate the optimal number of checkpoints, insert these values in Equation (2), and carry out the feasibility test.

(1) Application-level speed scaling: here all tasks have the same speed f^* and $s(\tau_1) = s(\tau_2) = \ldots = s(\tau_n) = f^*$, where $f^* \in \{f_1, f_2, \ldots, f_l\}$. Then Equation (2) is simplified as:

$$\begin{aligned} R_i^{(j+1)} =& [(E_i + kE_i/(m_i^*+1))/f^* + k(C_s + C_r) + m_i^* C_s] \\ &+ \sum_{h=1}^{i-1} \lceil R_i^{(j)} / T_h \rceil \; [(E_h + kE_h/(m_h^*+1))/f^* + k(C_s + C_r) + m_h^* C + t_{ss}] \end{aligned}$$

For each given speed f^*, in order to minimize all response times, we must have: $m_i^* = \max(\left\| \sqrt{kE_i/(f^* C_s)} - 1 \right\|, 0), \; 1 \le i \le n$. The iterative method described in Section 3.1 can be used here. To examine the feasibility for each task, all l possible speeds have to be examined. The lowest speed that satisfies the timing constraints is selected to minimize energy consumption.

(2) Task-level speed scaling: to obtain an optimal solution, a straightforward solution is to use an exhaustive search method. Since each task can be run at l speeds, there are l^n possible speed combinations for n tasks. Given a speed assignment, in order to minimize all response times, we must have: $m_i^* = \max(\left\| \sqrt{kE_i/(s(\tau_i)C_s)} - 1 \right\|, 0), \; 1 \le i \le n$. The feasibility test is performed according to Equation (2). Meanwhile, the energy consumption is calculated from Equation (3). A speed combination that satisfies the timing constraints with the minimum energy consumption is chosen as the optimal solution.

4.2 Hyperperiod-oriented fault-tolerance with DVS

Let $F_j = E_j/[s(\tau_j)(m_j+1)]$. The worst-case response time for task τ_i can be expressed as:

$$\begin{aligned} R_i =& [E_i/s(\tau_i) + m_i C_s] + \sum_{h=1}^{i-1} \lceil R_i / T_h \rceil \; [E_h/s(\tau_h) + m_h C_s + t_{ss}] \\ &+ k(C_s + C_r) + k \max_{1 \le j \le i} \{F_j\} \end{aligned} \quad (4)$$

The total energy consumption during one hyperperiod is expressed as:

$$\begin{aligned} Total\_eng =& \sum_{i=1}^{n} Ht/T_i [Eng(E_i, s(\tau_i)) + m_i Eng\_C_s + Eng\_ss] \\ &+ k(Eng\_C_s + Eng\_C_r) + kEng(F^* s(\tau^*), s(\tau^*)) \end{aligned} \quad (5)$$

Here τ^* is the task with the longest checkpointing interval, F^* represents its checkpointing interval and $s(\tau^*)$ represents its corresponding speed assignment.

(1) Application-level speed scaling: here all tasks have the same speed f^* and $s(\tau_1) = s(\tau_2) = \ldots = s(\tau_n) = f^*$, where $f^* \in \{f_1, f_2, \ldots, f_l\}$. Let $F_j = E_j/[f^*(m_j+1)]$. Then Equation (4) is simplified to:

$$R_i = [E_i/f^* + m_i C_s] + \sum_{h=1}^{i-1} \lceil R_i/T_h \rceil \; [E_h/f^* + m_h C_s + t_{ss}] + k(C_s + C_r) + k \max_{1 \le j \le i} \{F_j\}$$

For each given speed f^*, we examine the feasibility of the task set using the method in Section 3.2. If it is schedulable, the corresponding number of checkpoints for each task can be obtained. The energy consumption is calculated from Equation (5). The lowest speed that satisfies the timing constraints is selected to minimize energy consumption.

(2) Task-level speed scaling: to obtain an optimal solution, we use an exhaustive method and enumerate l^n speed combinations. For each speed combination, the feasibility test is performed according to Equation (4). Energy consumption is calculated from Equation (5). The speed combination that satisfies the timing constraints with the minimum energy

```
Procedure Init($\Gamma$)
Input: Task set $\Gamma$ and processor speeds $f_1, f_2, \ldots, f_l$
Ouput: Initial chromosome population $\Omega$ with a size of $P$
(1) Find the lowest speed $f^*$ which makes the task set schedulable;
(2) Generate one chromosome $\alpha_0=(f^*, f^*, \ldots, f^*)$
(2) Apply random mechanism to generate the other $(P - 1)$ chromosomes

Procedure GA($\Omega$)
Input: Initial chromosome population: $\Omega = \{\alpha_i | \alpha_i = (v_{i1}, v_{i2}, \ldots, v_{in}), 1 \le i \le P\}$
Ouput: chromosome $\alpha^*=(f_1^*, f_2^*, \ldots, f_n^*)$ which makes the task set schedulable and minimizes energy consumption
(1) while number of generations not exhausted do
(2)     for $j$ =1 to PopulationSize do
(3)         Select two chromosomes with the highest fitness values, apply the crossover operator randomly, generate two children;
(4)         Apply mutation to two children randomly, update their fitness values;
(5)     endfor
(6) endwhile
(7) Report the best chromosome as the final solution
```

Figure 2. Heuristic search based on GA.

consumption is chosen as the optimal solution.

4.3 Heuristic method based on a genetic algorithm

The exhaustive method for task-level speed scaling is very time-consuming, especially when the size of the task set or the number of processor speeds is large. We present here a heuristic procedure based on genetic algorithms.

The heuristic procedure is divided into two stages: application-level population generation, and task-level heuristic search. The procedure is described in Figure 2. Each chromosome α_i is an n-dimensional vector $(v_{i1}, v_{i2}, \ldots, v_{in})$, where n is the number of tasks and v_{ij} is the corresponding speed for task τ_j. Furthermore, α_i is *viable* only if the task set can be scheduled under the corresponding speed assignment. Procedure *Init*(Γ) initializes the search space (chromosome population). One chromosome is initially generated based on the application-level speed scaling. This is to ensure that the initial population always includes a schedulable solution if such a solution exists. The other chromosomes are generated randomly. The initial population Ω is composed of these chromosomes. Procedure *GA*(Ω) applies crossover and mutation operators to Ω based on the fitness values. The operations are repeated for a predefined number of generations Q. The fitness value $fit(\alpha_i)$ is calculated as follows: (1) If α_i is not viable: $fit(\alpha_i)= rand(\ )$, where $rand(\ )$ is a uniform random function that returns a value between 0 and 1; (2) If α_i is viable: $fit(\alpha_i) = 0.6 + 0.4 \times B/Energy(\alpha_i)$, where B is a constant and $Energy(\alpha_i)$ is the energy consumption for the task set under chromosome α_i.

Since the fitness value of a viable chromosome is always greater than 0.6, and the fitness value of a chromosome that is not viable is a random number whose probability is between 0 and 1 with a uniform probability distribution, there is a higher probability that a viable chromosome is selected to generate children. Furthermore, a chromosome with low-energy consumption has a high fitness value, which makes it more likely to be selected.

The mutation and crossover operators used in the procedure are defined as follows:

(1) *Crossover*: find an index randomly; then one child keeps the information of its parent to the left of the index and fills the right with the other parent chromosome, and the other child keeps the information of its parent to the right of the index and fills the left with the other parent chromosome.

(2) *Mutation*: choose a certain number of bits from two children randomly and replace them with different information.

The complexity of this heuristic method is linear in the number of generations Q and the population size P.

5. Simulation results

We next compare the performance of the proposed scheme with the VSLP technique proposed in [2]. We also compare our approach with a fault-tolerant scheme that does not consider energy.

We use the following notation to refer to the different variants of our scheme: 1) JFTC: job-oriented fault tolerance under constant speed; 2) JFTA: job-oriented fault tolerance with application-level speed scaling; 3) JFTT: job-oriented fault tolerance with task-level speed scaling; 4) HFTC: hyperperiod-oriented fault tolerance under constant speed; 5) HFTA: hyperperiod-oriented fault tolerance with application-level speed scaling; 6) HFTT: hyperperiod-oriented fault-tolerance with task-level speed scaling.

We choose two low-power embedded processors for our experiments: Intel XScale PXA260 [17] and Transmeta Crusoe [18]. The relevant parameters for these processors are listed in Table 1.

We evaluate our schemes on three real-life task sets [1]. These task sets include a computer numerical control (CNC) task set, an inertial navigation system (INS) task set, and a generic aviation platform (GAP) task set, respectively. The task execution times (provided in μs in the literature) are assumed for a nominal CPU frequency of 200 MHz.

Based on the discussion in Section 2, we assume that the checkpoint size is 5 KB, and that checkpoint data is saved in DRAM. Based on the typical access speeds of DRAM, the time to read or write a checkpoint of size 5 KB is assumed to be 0.4 ms. We choose a power consumption value of 400mW for the DRAM [19]. Hence the energy consumption for saving or retrieving a checkpoint is 160 μJ. In addition, based on data provided in the literature in [14], we assume that a single DVS transition takes 100 μs, and consumes 30 μJ.

Since the number of tasks for CNC and INS is relatively small, the simulation results for CNC and INS are obtained using the exhaustive search method. The simulation results for GAP are obtained using the heuristic method.

Simulation results on JFTT for the Intel XScale processor are shown in Table 2. The last two columns of the table show the energy saving of JFTT compared to VSLP and JFTC, respectively. In the table, “NF” denotes that the task set cannot be feasibly scheduled, $E_{13}=(E_1-E_3)/E_1\times100\%$, and $E_{23}=(E_2-E_3)/E_2\times100\%$. DVS is not effective for CNC because the voltage switching time is comparable to its task execution times. For INS and GAP, whose tasks have longer execution times compared to the checkpointing cost, JFTT outperforms VSLP. For INS with k = 2, VSLP is unfeasible but JFTT is still feasible. The performance for JFTC and JFTT are comparable. This is because JFTT has to often run at the highest processor speed to ensure timely task completion.

Intel XScale PXA260			Transmeta Crusoe		
CPU Frequency (MHz)	Voltage (V)	CPU Power (mW)	CPU Frequency (MHz)	Voltage (V)	CPU Power (W)
200	1.0	178	300	1.2	1.3
300	1.1	283	400	1.225	1.9
400	1.3	411	533	1.35	3.0
			600	1.5	4.2
			667	1.6	5.3

Table 1. Processor frequencies, voltages and power [19, 20].

Task Set	k	VSLP: E_1(mJ)	JFTC: E_2(mJ)	JFTT: E_3(mJ)	E_{13}	E_{23}
CNC	1	NF	25.1	25.1	–	0
INS	1	1830.8	1659.6	1657.6	9.5	0.1
	2	NF	1993.9	1993.9	–	0
GAP	1	40433.3	34758.5	34120.4	15.6	1.8

Table 2. JFTT for Intel XScale.

Task Set	k	VSLP: E_1(mJ)	JFTC: E_2(mJ)	JFTT: E_3(mJ)	E_{13}	E_{23}
CNC	1	NF	193.9	193.9	–	0
	2	NF	265.2	265.2	–	0
	3	NF	NF	NF	–	0
INS	1	8205.2	11853.5	5895.2	28.2	50.3
	2	15076.4	12633.0	7837.3	48.0	38.0
	3	NF	14802.8	14802.8	–	0
GAP	1	208334.5	239392.8	135858.2	34.8	43.2
	2	463647.2	274543.1	177580.4	61.7	35.3
	3	NF	298035.2	297312.7	–	0.2
	4	NF	338236.6	338236.6	–	0

Table 3. JFTT for Transmeta Crusoe.

Task Set	k	VSLP: E_1(mJ)	HFTC: E_2(mJ)	HFTT: E_3(mJ)	E_{13}	E_{23}
CNC	1	NF	98.2	98.2	–	0
	2	NF	172.6	172.6	–	0
	3	NF	NF	NF	–	0
INS	1	8233.2	7184.1	4010.4	51.3	44.1
	2	15147.6	7243.7	4032.6	73.4	44.3
	3	NF	7308.7	4208.7	–	42.4
	4	NF	7382.8	4672.5	–	36.7
	5	NF	7424.6	5326.3	–	28.3
	6	NF	7517.3	7517.3	–	0
GAP	1	208314.0	159429.2	90185.8	56.7	43.4
	2	463715.6	159443.7	98305.8	78.8	38.3
	3	NF	159995.1	109918.4	–	31.3
	4	NF	160003.2	126734.2	–	20.8

Table 4. HFTT for Transmeta Crusoe.

The simulation results for JFTT for the Transmeta Crusoe processor are shown in Table 3. Compared to VSLP, JFTT saves more energy when both schemes are feasible. For example, the energy saving for GAP with $k = 2$ is as high as 61.7%. JFTT can also tolerate more faults. For example, JFTT can tolerate 3 faults for INS and 4 faults for GAP, while VSLP can tolerate only 2 faults for INS and 2 faults for GAP. Next we compare JFTT and JFTC. As expected, JFTT saves more energy when the number of faults to be tolerated is small. For example, JFTT saves 43.2% energy over JFTC for GAP with $k = 1$, but it saves only 0.2% energy with $k = 3$.

Finally, the simulation results for HFTT for the Transmeta Crusoe processor are shown in Table 4. HFTT saves as much as 78.8% in energy over VSLP, and as much as 44.3% over HFTC.

6. Conclusions

We have shown how a combination of checkpointing and DVS can be used in real-time embedded systems. We have presented feasibility-of-scheduling tests for checkpointing schemes under both constant processor speed and variable processor speed. A heuristic method has been proposed to reduce the computational complexity for voltage scaling. We have presented simulation results for two commercial embedded processors using real-time benchmark task sets.

References

[1] Y. Shin and K. Choi, “Power conscious fixed priority scheduling for hard real-time systems”, *Proc. DAC*, pp. 134-139, 1999.

[2] G. Quan and X. Hu, “Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors”, *Proc. DAC*, pp. 828-833, 2001.

[3] K. G. Shin and Y.-H. Lee, “Error detection process—Model, design and its impact on computer performance”, *IEEE Trans. Computers*, vol. C-33, pp. 529-540, Jun. 1984.

[4] A. Ziv and J. Bruck, “An on-line algorithm for checkpoint placement”, *IEEE Trans. Computer*s, vol. 46, no. 9, pp. 976-985, Sep. 1997.

[5] S. W. Kwak et al, “An optimal checkpointing-strategy for real-time control systems under transient faults”, *IEEE Trans. Reliability*, vol. 50, no. 3, pp. 293-301, Sep. 2001.

[6] Y. Zhang and K. Chakrabarty, “Energy-aware adaptive checkpointing in embedded real-time systems”, *Proc. DATE*, pp. 918-923, 2003.

[7] R. Melhem et al, “The interplay of power management and fault recovery in real-time systems", to appear in *IEEE Trans. Computers*, 2003. Available online at:
http://www.cs.pitt.edu/PARTS/papers/ieeetc_03.pdf.

[8] Y. Zhang et al, “Energy-aware fault tolerance in fixed-priority real-time embedded systems”, *Proc. ICCAD*, pp. 209-214, 2003.

[9] E. N. Elnozahy et al, “The performance of consistent checkpointing”, *Proc. Reliable Distributed Systems*, pp. 39-47, 1992.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, CA, 2002.

[11] J. S. Plank et al, “Libckpt: Transparent checkpointing under Unix”, *Proc. Usenix Tech. Conf.*, pp. 213-223, 1995.

[12] C.-Y. Lin et al, “A checkpointing tool for Palm operating system”, *Proc. DSN*, pp. 71-76, 2001.

[13] A. Campbell et al, “Single event upset rates in space”, *IEEE Trans. Nuclear Science*, vol. 39, pp. 1828-1835, Dec. 1992.

[14] D. Grunwald et al., “Policies for dynamic clock scheduling”, *Proc. Symp. OSDI*, pp. 73-86, 2000.

[15] W. Ye et al, “The design and use of SimplePower: A cycle-accurate energy estimation tool”, *Proc. DAC*, pp. 340-345, 2000.

[16] J. W. Liu, *Real-Time Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[17] Intel® PXA26x Processor Family Electrical, Mechanical, and Thermal Specification Datasheet: http://developer.intel.com.

[18] Transmeta LongRun Power Management - Dynamic Power Management for Crusoe Processors: http://www.transmeta.com.

[19] T. Simunic et al, “Event-driven power management”, *IEEE Trans. CAD*, vol. 20, pp. 840-857, July 2001.